19. Diskutieren sie die beiden im PS zur Frage “Wie viele Menschen müssen in einem Raum sein, damit die Wahrscheinlichkeit grösser als 0.5 ist, dass eine der Personen an einem vorgegebenen Tag Geburtstag hat?” besprochenen Lösungen, und insbesondere, welche der beiden unter welchen Umständen korrekt ist.

auf die 253 kommt durch die zuvor besprochene Berechnung unter der Annahme, dass jede Person für jeden Tag dieselbe WSK hat an diesem Geboren zu sein, s.h. Es können mehrere Personen am selben Tag Geboren sein, es können aber auch alle an unterschiedlichen Geboren sein.
Rechnet man unter dieser Annahme den Wert aus braucht man wie vorgestellt 253 Personen um eine WSK von 50% zu erhalten.

Geht man nun allerdings davon aus, dass keine zwei Personen am gleichen Tag Geboren sind entspricht die WSK direkt dem verhältnis zwischen der Anzahl der Personen und der Anzahl an Tagen im Jahr. Gibt es nun z.b.: 365 Personen und es sind alle an Unterschiedlichen Tagen Geboren ist die WSK, dass eine Personen an einem bestimmtem zuvor gewählten Tag Geboren ist 100% da alle möglichen Tage belegt wurden. Somit errechnet sich die WSK unter dieser Annahme einfach durch $p = Personen/Tage$
haben wir also $0.5 = n/365$ formen wir um auf $0.5 * 365 = n$ haben wir $n = 182.5$ also aufgerundet 183.

20. **HÜ10**: Leiten sie die angegebenen Mengen von Hash-Werten her, die für die beiden Angriffsvarianten notwendig sind (2^m vs. $2^{m/2}$).

    2^m beim Suchen nach einem bestimmten M' für Welches H(M) = H(M') wenn M gegeben ist: Um Tatsächlich mit hoher WSK ein zweites M' zu finden muss man alle anderen Hash Werte durchgehen, schließlich kann man aus H(M) keine hilfreiche Aussagen über M treffen und demnach auch keine über ein potentielles M'. Bei einem m bit Hash ist logischerweise die Anzahl aller Hashes 2^m.

$2^{\frac{m}{2}}$ beim Suchen nach zwei M und M' wo keines der beiden gegeben ist, für welche H(M) = H(M'). Dieses Problem gleicht dem Geburtstagsparadoxon, die nötigen M die man produzieren muss um ein passendes Paar zu finden sind wesentlich kleiner, da für jedes zusätzliche M nicht nur eine weitere Vergleichsmöglichkeit hinzugefügt wird, wie bei der anderen Version des Problems, sondern jedes bereits bekannte M mit jedem anderen inkl. dem neuen verglichen werden kann. s.h. pro neuem M gibt es n-1 neue Vergleichspaare, somit haben wir bei n Ms $\frac{n*(n-1)}{2}$ Vergleichspaare (/2 da M1 und M2 nur einmal verglichen werden nicht M1 mit M2 und M2 mit M1 etc.)

Setzt man nun $2^{m/2}$ ein kommt man auf $\frac{2^{m/2}*(2^{m/2}-1)}{2}$, wenn wir das vereinfachen kommen wir auf $\frac{2^m-2^{m/2}}{2}$ in den meisten Fällen ist $2^{m/2}$ vernachlässigbar klein und somit kommen wir ungefähr auf $2^m/2$, s.h. dass die WSK ein paar zu finden ab $2^{m/2}$ ungefähr 50% beträgt, wie beim Geburtstagsparadoxon kann man um diesen Wert davon ausgehen früher oder später ein paar zu finden.

HÜ2: Erklären sie detailliert die folgenden Fragen:

- An welcher Stelle des beschriebenen Angriffs ist tatsächlich die Bedingung

$$V_x = E_x \quad \text{und} \quad S_x = D_x$$

für den Erfolg notwendig ? Warum funktioniert das ohne diese Bedingung nicht ?

- Warum kann bei einer Unterschriftsleistung bei der vor der Unterschrift One-Way Hash Funktionen eingesetzt werden dieser Angriff vereitelt werden ? In welchem Schritt ?

21.

1.) Warum bzw. wo muss gelten, dass Verschlüsseln und Verifikation bzw Signieren und Entschlüsseln Equivalent sind damit der Angriff funktioniert.

Die ursprüngliche Nachricht von Alice and Bob ist $E_B(S_A(m))$ diese wird von Mallory abgefangen und dann an Bob geschickt, unter der Behauptung sie wäre von Mallory.

$$V_M(D_B(E_B(S_A(m)))) = E_M(S_A(m))$$

Hier entschlüsselt Bob die Nachricht, dann, im Versuch Mallorys Signatur zu verifizieren, verschlüsselt er diese Mit Mallorys Public key da Vx = Ex. Die Daten die nun vorliegen können von Mallory ganz einfach mit seinem eigenen Private key und Alices Public key entschlüsselt werden. Die Einzige Hürde (Die vorige Verschlüsselung mit Bobs Public key) fällt hier weg.

Nachdem Bob dies dem Protokoll entsprechend an Mallory zurück sendet $E_M(S_B(E_M(S_A(m))))$ steht die Nachricht zur Verfügung.

Der Schritt in dem die Daten für Mallory lesbar werden funktioniert nur, da Die von Bob versuchte Verifikation (Vx) der Verschlüsselung mit Mallorys Public key entspricht (Ex) wären diese Operationen nicht equivalent würde das ganze nicht funktionieren, da Mallory ja nier tatsächlich die Nachricht signiert hat.

22. Erklären sie, warum die Verwendung von Hashfunktionen im Kontext mit digitalen Signaturen die Protokollattacke gegen digitale Empfangsbestätigungen verunmöglicht. Bedenken sie dabei insbesondere, dass in diesem Fall NachrichtenHash und Nachricht übermittelt werden.
    Nutzt man vor der Signatur bei der Empfangsbestätigung ein One-Way Hash wird der Angriff in dem Schritt vereitelt in dem Bob die Empfangsbestätigung sendet, sobald die Daten durch das One-Way Hash gehen ist die eigentliche Nachricht verloren. Hier erhält Mallory die nun für ihn Lesbare Nachricht als teil der Empfangsbestätigung welche so aussieht: $E_M(S_B(E_M(S_A(m))))$
    Hätte Bob hier aber eine One-Way Hash Funktion eingesetzt sehe das Ganze so aus: $E_M(S_B(H(E_M(S_A(m)))))$ alles was sich in dieser Hash Funktion befindet ist nicht entschlüsselbar, demnach könnte Mallory, falls er m schon kennen würde, zwar bestätigen, dass Bob auch m erhalten hat. Er kann allerdings nicht aus $H(E_M(S_A(m)))$ m herleiten. Somit ist der Angriff verhindert.
23. Führen sie eine Geburtstagsattacke (beschrieben auf S. 87 der VO-Slides) durch: Erstellen sie zwei semantisch unterschiedliche Dokumente (wie im besprochenen Beispiel die beiden unterschiedlichen Mietvertr¨age) im Format ihrer Wahl (Word, Postscript, etc.), und modifizieren sie beide Varianten Semantik-erhaltend automatisiert (beschreiben sie das detailliert, wie sie dabei vorgehen), bis sie auf zwei Varianten mit dem gleichen hash-Wert treffen (verwenden sie als Hash-Wert einen Teil des Outputs der eigentlich obsoleten Hashfuntion MD-5).

```
def birthday_attack(file1, file2, tries):
    hash_changes1 = generate_hash_change_list(file1, tries)
    hash_changes2 = generate_hash_change_list(file2, tries)

    for i in range(tries):
        for j in range(tries):
            if hash_changes1[i][0] == hash_changes2[j][0]:
                print(f'Collision found')
                return hash_changes1[i][1], hash_changes2[j][1]

    print('No collision found')

birthday_attack('fair.docx', 'unfair.docx', 10000)
```

Hier generieren wir zwei Arrays, im Format `[[hashwert, änderungen], [hashwert, änderungen], ...]` wobei "änderung" wiederum ein array ist welches die Änderungen am file darstellt (also font, italics, ...) somit haben wir "tries" viele Versionen beider dateien welche dann auf eine Hash-Kollision überprüft werden.

Diese Liste an Hashwerten und Änderungen generieren wir folgendermaßen:

```
def generate_hash_change_list(file_path, i):
    hash_changes = []

    for _ in range(i):
        changes = modify_docx(file_path)
        hash = hash_file(file_path)
        hash_changes.append((hash, changes))

    return hash_changes
```

Wir nehmen jeweils den path, ändern das docx file zufällig und berechnen danach den hashwert des veränderten files. Diese packen wir für jede Variation in die Liste und geben nachdem wir fertig sind die gesamte Liste aus.

Der Hashwert wird wie üblich berechnet:

```
def hash_file(filename):
    hasher = hashlib.md5()

    with open(filename, 'rb') as file:
        chunk = 0

        while chunk != b'':
            chunk = file.read(8192)
            hasher.update(chunk)

    return hasher.hexdigest()
```

Die Variationen enstehen folgendermaßen:

```
def modify_docx(file_path):
    '''generates a random combination of font size, bold, italic, and underline for each
paragraph in the document'''
    doc = Document(file_path)
```

```
    changes = []

    for paragraph in doc.paragraphs:
        modif = []
        modif.append(random.randint(1, 61))
        modif.append(random.choice([False, True]))
        modif.append(random.choice([False, True]))
        modif.append(random.choice([False, True]))

        for run in paragraph.runs:
            run.font.size = Pt(modif[0])
            run.font.bold = modif[1]
            run.font.italic = modif[2]
            run.font.underline = modif[3]
        changes.append(modif)

    doc.save(file_path)
    return changes
```

Hier gehen wir Absatz für Absatz durch und generieren zufällige Werte für die Schriftgröße, und andere Texteigenschaften. Die Werte jedes Absatzes bekommen einen eigenen Eintrag in der changes Liste. Diese geben wir dann nach speichern des docx aus. Nachdem es hier 488 mögliche permutationen pro Absatz und laut docx Modul pro Dokument ~30 Absätze gibt haben wir ~488^30 Variationen pro Dokument, also an sich mehr als genug um eine Kollision finden zu können.

Dennoch hat sich in meinen Tests bis jetzt noch keine Kollision ergeben, aber ab 10000 tries wird die Zeit um alle Variationen zu berechnen einfach etwas zu lang.

Repo

- Github Repository